Adaptive atomic capture of multiple molecules

Marin Bertier, Marko Obrovac, Cédric Tedeschi

Adaptive Atomic Capture of Multiple Molecules

Marin Bertier[a,b,d], Marko Obrovac[a,c,d,*], Cédric Tedeschi[a,c,d]

[a] *IRISA, France*
[b] *INSA de Rennes, France*
[c] *Université de Rennes 1, France*
[d] *INRIA Rennes - Bretagne Atlantique, France*

Abstract

Facing the scale, heterogeneity and dynamics of the global computing platform emerging on top of the Internet, autonomic computing has been raised recently as one of the top challenges of computer science research. Such a paradigm calls for alternative programming abstractions, able to express autonomic behaviours. In this quest, nature-inspired analogies regained a lot of interest. More specifically, the chemical programming paradigm, which envisions a program's execution as a succession of reactions between molecules representing data to produce a result, has been shown to provide some adequate abstractions for the high-level specification of autonomic systems.

However, conceiving a runtime able to run such a model over large-scale platforms raises several problems, hindering this paradigm to be actually leveraged. Among them, the atomic capture of multiple molecules participating in concurrent reactions is one of the most significant.

In this paper, we propose a protocol for the atomic capture of these molecules distributed and evolving over a large-scale platform. As the density of potential reactions has a significant impact on the liveness and efficiency of such a capture, the protocol proposed is made up of two sub-protocols, each of them aimed at addressing different levels of densities of potential reactions in the solution. While the decision to choose one or the other is local to each node participating in a program's execution, a global coherent behaviour is obtained. We also give an overview of the course of execution when a program contains multiple rules and provide a rule-changing mechanism. The proof of correctness, as well as intensive simulation results showing the efficiency and limited overhead of the protocol are given.

*Corresponding author
Email addresses: marin.bertier@irisa.fr (Marin Bertier), marko.obrovac@inria.fr (Marko Obrovac), cedric.tedeschi@irisa.fr (Cédric Tedeschi)

1. Introduction

The amount of computing technologies surrounding us seems to grow everyday. These technologies provide applications, or *services*, that cover a growing part of our life. The underlying infrastructures (*e.g., the Internet*) make it possible to aggregate a virtually infinitely large set of these independent computing devices. Due to its scale and its highly dynamic and heterogeneous nature, leveraging the capabilities of such a platform is a quite tedious experience.

Enhancing these omnipresent computing systems around us with autonomic behaviours has been recently raised as one of the top challenges in computer science research. As for instance put by Parashar and Hariri in [23], *autonomic computing* is a paradigm consisting in building systems that can "*manage themselves in accordance with high-level guidance from humans.*" In other words, according to this paradigm, humans should simply define these *rules* initially. Then the system can virtually run indefinitely, adhering to these high-level, technical-detail-free rules, regardless of the conditions in the underlying platforms.

The implementation of such a system offers several challenges that cannot be tackled at once. A prerequisite is to separate the development of the low-level machinery from the definition of adequate programming abstractions, allowing this high-level human guidance. This situation advocates the use of declarative programming [18], whose goal is to separate the logic of a computation (*"what we want to do"*) from its control (*"how to achieve it"*). More precisely, while the "what" is to be defined by the programmer, the "how" becomes for them implicit, hidden in the system. In particular, rule-based programming, where this logic is expressed as a set of high-level *rules* allows to hide the intrinsic difficulties of the parallelism and distribution of the runtime from the programmer. Recently, some work has gone into showing how to concretely apply rule-based programming to the specification of distributed systems. For instance, in [12], it has been shown how communication protocols and peer-to-peer applications can be specified using a rule-based language. In [1], the same programming style is applied to web-based data management. On the computing side, rule-based programming was also used as a building block for workflow management systems [33, 15].

Also, nature appears to be a great source of inspiration. Biological and chemical systems[1] appear as analogies worth to be explored [2, 13, 32]. More concretely, the chemical programming model, which is both rule-based and nature-inspired, seems to be one of the most promising paradigms to be studied in such a quest [6, 11, 20, 22]. Combining rule-based programming with a nature-inspired paradigm results in a model which has got a high level of abstraction, whose paradigm is implicitly parallel and non-deterministic. The user is, thus, able to provide high-level rules, on the basis of which the system is able to

[1] The autonomic computing paradigm was initially proposed by analogy with the autonomic nervous system.

manage itself due to the implicit parallelism and non-determinism.

Metaphorically speaking, in such a model, a program is envisioned as a *chemical solution* where molecules of data float and react according to some *reaction rules* specifying the program, to produce new data (the products of reactions). Formally speaking, these artificial chemistries [10] rely on *concurrent multiset rewriting* : the solution is a multiset of molecules, and reactions are rewriting rules to be applied on it. At run time, reactions can be triggered concurrently. The exact degree of parallelism may vary from one model to another. For instance, membrane computing [24] is a branch of chemical computing focusing on the organisation of the solution in sub-solutions, in which it is assumed that, at each step, the set of reacting molecules is maximised. However, this is not the case for instance for the runtime of the HOCL language [5], whose parallelism degree remains unspecified, and is left to the implementer of the underlying runtime.

Once no more reactions are possible, the program is said to be *inert*. In this state, the solution is stable and contains the result of the computation.

While the chemical paradigm offers a promising way to specify autonomic systems, running these chemical specifications over distributed platforms is still a widely open issue. One of the most significant barriers to be lifted is inherent to concurrent rewriting and deals with the atomic capture of the different molecules satisfying a reaction. At run time, a molecule can potentially participate in several concurrent reactions. However, it should be ensured that it will participate in at most one. Otherwise, the logic of the program could be broken (as exemplified in Section 2).

Let us slightly refine the problem envisioned in this paper : we consider a chemical program made of a multiset of objects (molecules), and a set of rules to be applied concurrently on them. Both the objects and the rules are distributed over a set of nodes on which the program runs. Each node periodically tries to fetch molecules needed for the reactions it is trying to perform. As several molecules can satisfy the pattern and conditions of several reactions performed concurrently by different nodes, the same molecule can be requested by several nodes at the same time, inevitably leading to conflicts. Mutual exclusion on the molecules is thus mandatory. Although our problem resembles the classic resource allocation problem [16], it differs from it in several aspects. Firstly, the molecules are interchangeable to some extent. The requested molecules must match a pattern defined in the reaction rule a node wants to perform; if two molecules, say A and B, both match the rule's pattern, any of the two may be used in the actual reaction. Then, we differentiate two processes which are :

1. finding molecules matching a pattern (achieved by a *discovery protocol*);
2. obtaining them to perform reactions (achieved by a *capture protocol*).

Consequently, if one node cannot manage to grab some specific molecules, it will switch to another set of molecules. We are not so much interested in avoiding one node's starvation as in the liveness of the system itself : *some* node should be able to perform one reaction in a finite time in order to move the computation forward.

Secondly — and following the previous point — the platform envisioned is at large scale, and the resources dispatched over the nodes are dynamic : molecules are deleted when they react, and new ones are created. Likewise, the number of resources/molecules (and of possible reactions) will fluctuate over time, influencing the design of the capture protocol. Bear in mind that once the holder of a matching molecule is located, the scale of the network is of less importance, since only the nodes requesting the molecules and their holders are involved in the capture protocol.

To sum up, our objective is to define a protocol for the atomic capture of multiple molecules that dynamically and efficiently adapts to the density of potential reactions in the system.

Contribution. Our contribution is a distributed protocol combining two sub-protocols inspired by previous works on distributed resource allocation, and adapted to the distributed runtime of *chemical* programs. The first sub-protocol, referred to as the *optimistic* one, assumes that the number of molecules satisfying some reaction's pattern and condition is high, so only few conflicts for molecules will arise, nodes being likely to be able to grab distinct sets of molecules. While this protocol is simple, fast, and has a limited communication overhead, it does not ensure liveness when the number of conflicts increases. The second one, called *pessimistic*, slower, and more costly in terms of communication, ensures liveness in the presence of an arbitrary number of conflicts. Switching from one protocol to the other is achieved in a scalable, distributed fashion and is based on local success histories in grabbing molecules. Furthermore, we analyse chemical programs containing multiple rules and the possible input/output dependencies they might have and propose a rule-changing mechanism instructing nodes as to which rule to execute. A proof of the protocol's correctness is given, and its efficiency is discussed through a set of simulation results. Note that the bare protocol and preliminary results have previously been published in [7], while this paper provides an in-depth description of the complete algorithm and an analysis of all of its features.

Organisation of the Paper. The next section presents the chemical programming paradigm in more detail, highlights the need for the atomic capture, and describes the system model used throughout the paper. Section 3 details the sub-protocols, their coexistence, and the switch from one to the other. It also proposes a communication-minimisation scheme when the number of conflicts is high as well as the aforementioned rule-changing mechanism. Proofs of safety and liveness are also given for the complete protocol. Section 4 presents the simulation results and discusses the efficiency and overhead of the protocol. Related works, both in the chemical programming and the distributed systems areas, are presented in Section 5. Section 6 concludes.

2. Preliminaries

Different systems require different algorithms for performing atomic operations varying in complexity. This section describes the programming and system

models which compose the required conditions for the proposed protocol.

2.1. Concurrent Multiset Rewriting

Concurrent multiset rewriting, as formalised by the γ-calculus [4], was put into practice by the Higher-Order Chemical Language (HOCL) [5]. As suggested by the chemical metaphor, in such a language, data are molecules floating in a solution. At runtime, they are consumed according to some reaction rules, *i.e.* the program, producing new molecules, *i.e.* the resulting data. These reactions take place in an implicitly parallel and autonomous way, until no more reactions are possible, a state referred to as *inertia*. More specifically related to HOCL, which includes the *higher order*, rules can themselves be put in the multiset, and be consumed and/or produced in a reaction. As our focus in this paper is the actual capture of molecules in a large-scale system, the higher order is not our primary concern. Hence, for the sake of simplicity, we illustrate the paradigm in its first-order version. In HOCL, a rule takes the form:

$$\textbf{replace } P \textbf{ by } M \textbf{ if } V$$

It consumes a set of molecules satisfying the pattern P and the condition V, and produces a set of molecules M. We want to emphasise here that consumption is the only possible transition in the state of a molecule: once it has been consumed, it vanishes from the multiset entirely; meaning molecules are objects that are only created and deleted, never updated nor recreated. For the sake of illustration, let us consider the following chemical program made up of two rules applied on a multiset of strings, that counts the aggregated number of characters in words with two or more letters:

$$\begin{aligned}
&\textbf{let } \mathit{count} &&= \textbf{replace } s :: \mathit{string} \textbf{ by } \mathit{len}(s) \textbf{ if } \mathit{len}(s) \geq 2 \textbf{ in}\\
&\textbf{let } \mathit{aggregate} &&= \textbf{replace } x :: \mathit{int}, y :: \mathit{int} \textbf{ by } x + y \textbf{ in}\\
&\quad \langle\ \text{"}\mathit{maecenas}\text{"}, \text{"}\mathit{ligula}\text{"}, \text{"}\mathit{massa}\text{"}, \text{"}\mathit{varius}\text{"}, \text{"}\mathit{a}\text{"}, \text{"}\mathit{semper}\text{"},\\
&\qquad\qquad \text{"}\mathit{congue}\text{"}, \text{"}\mathit{euismod}\text{"}, \text{"}\mathit{non}\text{"}, \text{"}\mathit{mi}\text{"}\ \rangle
\end{aligned}$$

The rule named *count* consumes a *string* element if it is composed of at least two characters, and introduces an integer representing its length into the solution. The *aggregate* rule consumes two integers to produce their sum. By its repeated execution, this rule aggregates the sums to produce the final number. At runtime, these rules are executed repeatedly and concurrently, the first one producing inputs for the second one. While the result of the computation is deterministic, the order of its execution is not. Only the mutual exclusion of reactions sharing some reactants, through the atomic capture of the reactants is implicitly required by the paradigm.

A possible execution is the following. Let us consider, *arbitrarily*, that the first rule is applied on the first three strings as represented above, and on the last one. The state of the multiset is then the following:

$$\langle\ \text{"}\mathit{varius}\text{"}, \text{"}\mathit{a}\text{"}, \text{"}\mathit{semper}\text{"}, \text{"}\mathit{congue}\text{"}, \text{"}\mathit{euismod}\text{"}, \text{"}\mathit{non}\text{"}, 8, 6, 5, 2\ \rangle.$$

Then, let us assume, still arbitrarily, that the *aggregate* rule is triggered three times on the previously introduced integers, producing their sum. Meanwhile, the remaining strings are scanned by the *count* rule. The multiset is then:

$$\langle\ 6, "a", 6, 6, 7, 3, 2, 21\ \rangle.$$

With the repeated application of the *aggregate* rule, inertia is reached ("a" satisfies neither of the two rules' conditions). The final solution is:

$$\langle\ "a", 51\ \rangle.$$

It is important to notice that the atomic capture is a fundamental condition. Let us simply assume that the same string was captured by different nodes running the *count* rule in parallel. Then, the count for a word would appear more than once in the solution, leading to an incorrect result.

The reader may have noted that the presented example is in fact a MapReduce-like [9] program, where the rule *count* provides the map function, while *aggregate* is the *reducer*. However, note that this example has not been given for comparison reasons, but in order to draw the reader's attention to: *(i)* the execution paths a chemical program might take; and *(ii)* the importance the atomic capture of molecules bears on the correctness of the execution. A comparison between the chemical programming model and MapReduce, or other paradigms used for parallel execution in HPC systems such as MPI [21], is not in order due to the difference in their target systems: while MapReduce and MPI target HPC systems with the aim of improving the execution performance, our intention is to use the chemical programming model for the coordination and management of large-scale, distributed systems, in an autonomic fashion, as for instance shown in detail in [6, 11, 20, 22].

2.2. System Model

We consider a distributed system $\mathbb{DS}$ consisting of n machines which communicate via message passing. They are interconnected in such a way that a message sent from a node can be delivered, in finite time, to any other node in $\mathbb{DS}$. Moreover, we suppose that a communication channel between any two given nodes is a FIFO queue — a message sent at time t is always delivered strictly before a message sent at time $t+\epsilon$. At large scale, such a fully-connected network can be built by relying on P2P systems, more specifically ones employing distributed hash table (DHT) communication protocols [26, 31]. They allow us to focus on the atomic capture of molecules without worrying about the underlying communications' details.

Data and Rules Dissemination. In the following, we assume data and rules have already been dispatched to the nodes. Again, any DHT algorithm or network topology may be used for this purpose. Even if the data and rules are initially held by a single external application, it can contact a node in the DHT and transfer it the chemical solution to be executed. The node which received the data scatters the molecules across the overlay according to the DHT's hash

function. Molecules are routed concurrently according to the DHT's routing scheme. The dissemination of rules can follow a similar pattern, or can be broadcast in the network. The only difference is that rules can be replicated on several nodes to satisfy an increased level of parallelism. A more accurate discussion of the rules' distribution falls out of the scope of this paper. In the following, we simply assume every rule of the program is present on all of the nodes in the system.

Discovery Protocol. In order for the reaction to happen, a suitable combination of molecules has to be found. While the details of this aspect are also abstracted out in the remainder of the paper, they deserve to be preliminarily discussed. The basic *lookup* mechanism offered by DHTs allows the retrieval of an object according to its (unique) identifier. Unlike the *exact match* functionality provided by DHTs, we require nodes to be able to find *some* molecule satisfying a pattern (*e.g.*, one *integer*) and condition (*e.g.*, *greater than* 3), as stated in Section 2.1. This can be achieved by the support of range queries on top of the overlay network, *i.e.* mechanisms to find some (at least one) molecules falling within a range, provided the molecules can be totally ordered on a (possibly complex, multi-dimensional) criterion, as for instance provided in [28]. This mechanism can be easily extended to support patterns and conditions involving several molecules. For instance, when trying to capture two molecules ordered in a specific way, a *rule translator* constructs the range query to be sent over the DHT based on the given rule and the first molecule obtained. If matching molecules are found, the capture protocol will be triggered.

Fault tolerance. DHT systems inherently provide fault-tolerance mechanisms. If nodes crash, leave or join, the properties of the communication pattern will be preserved. On top of that, we assume that there exists a higher-level resilience mechanism which prevents the loss of molecules, such as state machine replication [19, 29]. Each node replicates its complete state — the molecules and its current actions — across k neighbouring nodes, where the definition of a *neighbour* depends on the actual DHT scheme used. Thus, in case of its failure, one of its neighbours is able to assume its responsibilities and continue the computation.

3. Protocol

Here, the protocol in charge of the atomic capture of molecules is discussed. The protocol can run in two modes, based on two different sub-protocols: an *optimistic* and a *pessimistic* one. The former is a simplified sub-protocol which is employed while the number of possible reactions is high enough to render the possibility of conflicts insignificant. When the ratio between actual and possible reactions drops below a given threshold, the pessimistic sub-protocol is activated. While being the heavier of the two in terms of network traffic, this sub-protocol ensures the liveness of the system, even when an elevated number of nodes in it compete for the same subset of molecules.

3.1. Pessimistic Sub-protocol

To some extent similar to the three-phase commit protocol [30] used in database systems, this sub-protocol ensures that at least one node wanting to execute a reaction will succeed. The differences between the original protocol and our adaptation are discussed in Section 5. Molecule fetching is done in three phases — the *query*, *commitment*, and *fetch* phases — and involves at least two nodes — the node requesting the molecules, called *requester*, and at least one node holding the molecules, called *holder(s)*. Algorithms 1 and 2 represent the code run on these two entities, respectively, and Figure 1 delivers the time diagram of molecule fetching. Note that a node acts at times as a requester (when it executes rules), while at others it behaves as a holder (when it holds a molecule requested by another node).

When molecules suitable for a reaction have been found using the discovery protocol (line 1 in Algorithm 1), the query phase begins (line 10). The requester sends *QUERY* messages asynchronously to all of the holders to inform them it is interested in the molecule. Depending on their local states, each of the holders evaluates separately the received message (lines 1—13 in Algorithm 2) and replies with one of the following messages:

- *RESP_OK*: the requested molecule is available;
- *RESP_REMOVED*: the requested molecule no longer exists;
- *RESP_TAKEN*: the molecule has already been promised to another node.

Unless it received only *RESP_OK* messages, the requester aborts the fetch and sends *GIVE_UP* messages to holders, informing them it no longer intends to fetch their molecules (line 14 in Algorithm 1).

Following the query phase is the commitment phase, when the requester tries to secure its position by asking the guarantee from the holders that it will be able to fetch the molecules (line 19 in Algorithm 1). It does so using *COMMITMENT* messages. Upon its receipt, each holder sorts all of the requests received during the query phase (line 14 in Algorithm 2) according to the conflict resolution policy (described below). Holders reply, once again, with *RESP_OK*, *RESP_REMOVED* or *RESP_TAKEN* messages. A *RESP_OK* response represents a holder's commitment to deliver its molecule in the last phase. Thus, subsequent *QUERY* and *COMMITMENT* requests from other nodes will be resolved with a *RESP_TAKEN* message. Naturally, if a requester does not receive only *RESP_OK* responses to its *COMMITMENT* requests, it aborts the fetch with *GIVE_UP* messages. The holder then removes the requester from the list, in this way allowing others to fetch the molecule.

Finally, in the fetch phase, the requester issues *FETCH* messages, upon which holders transmit it the requested molecules using *RESP_MOLECULE* messages. From this point on, holders issue *RESP_REMOVED* messages to nodes requesting the molecule.

Conflict Resolution. Each of the holders individually decides to which requester a molecule will be given. Since we want at least one requester to be able to complete its combination of molecules, all holders apply the same conflict resolution scheme, based on the total order of requesters (lines 20—27 in Algorithm 2). Any total order scheme could be applied. We here detail a dynamic scheme based on load-balancing: each of the messages sent by requesters contains two fields — the requester's identifier and the number of reactions it has completed thus far. When two or more requesters are competing for the same molecule, holders give priority to the requester with the lowest number of reactions. In case of a dispute, the requester with a lower node identifier (ensured to be unique by the DHT's hash function) gets the molecule. Such a conflict resolution scheme promotes fairness while at the same time balancing the workload amongst nodes, seeing that the less reactions a node has done the greater the chances are for it to capture the molecules it needs for a reaction.

3.2. Optimistic Sub-protocol

When the probability of successful multiple concurrent reactions is high, the atomic fetch procedure can be relaxed and simplified by adopting a more optimistic approach. The optimistic sub-protocol requires only two phases — the *fetch* and the *notification* phases. Algorithm 3 describes the sub-protocol on the requesters' side, while Algorithm 4 describes it on the holders' side. The time diagram of the process of obtaining molecules is depicted in Figure 2.

Once a node acquires information about suitable candidates, it immediately starts the fetch phase (line 1 in Algorithm 3). It dispatches *FETCH* messages to the appropriate holders. As with the pessimistic sub-protocol, the holder can respond using one of the three previously described types of messages (*RESP_MOLECULE*, *RESP_TAKEN* and *RESP_REMOVED*) as shown in Algorithm 4. A holder that replied with a *RESP_MOLECULE* message, replies with *RESP_TAKEN* messages to subsequent requests until the requester either returns the molecule or notifies it a reaction took place.

If the requester acquires all of the molecules, the reaction is subsequently performed, and the requester sends out *REACTION* messages to holders to notify them the molecules are being consumed. This causes holders to reply with *RESP_REMOVED* messages to subsequent requests from other requesters. In case the requester received a *RESP_REMOVED* or a *RESP_TAKEN* message, it aborts the reaction and returns the obtained molecules by enclosing them in *GIVE_UP* messages, which allows holders to give them to others.

Conflict Resolution. Given the fact that the optimistic sub-protocol is designed to be executed by nodes in a highly *reactive* stage, there is no need for a strict conflict resolution policy. Instead, the node the request of which first reaches a holder obtains the desired molecule. However, the optimistic sub-protocol does not ensure that a reaction will be performed in case of a conflict. In the worst case, all attempts at fetching molecules might be aborted.

Algorithm 1: Pessimistic Sub-protocol — Requester.

```
1  on event combination found
2      QueryPhase(combination);
3  on event response received
4      if phase = query then
5          QueryPhaseResp(resp_mol);
6      else if phase = commitment then
7          CommitmentPhaseResp(resp_mol);
8      else if phase = fetch then
9          FetchPhaseResp(resp_mol);
10 begin QueryPhase(combination)
11     phase ⇐ query;
12     foreach molecule in combination do
13         dispatch QUERY(molecule);
14 begin QueryPhaseResp(resp_mol)
15     if resp_mol ≠ RESP_OK then
16         Abandon(combination);
17     else if all responses have arrived
       then
18         CommitmentPhase(combination);
19 begin CommitmentPhase(combination)
20     phase ⇐ commitment;
21     foreach molecule in combination do
22         dispatch COMMITMENT(molecule);
23 begin CommitmentPhaseResp(resp_mol)
24     if resp_mol ≠ RESP_OK then
25         Abandon(combination);
26     else if all responses have arrived
       then
27         FetchPhase(combination);
28 begin FetchPhase(combination)
29     phase ⇐ fetch;
30     foreach molecule in combination do
31         dispatch FETCH(molecule);
32 begin FetchPhaseResp(resp_mol)
33     add resp_mol to reaction_args;
34     if all responses have arrived then
35         Reaction(reaction_args);
36 begin Abandon(combination)
37     phase ⇐ none;
38     foreach molecule in combination do
39         dispatch GIVE_UP(molecule);
```

Algorithm 2: Pessimistic Sub-protocol — Holder.

```
1  on event message received
2      if message = GIVE_UP then
3          remove sender from
           molecule.list;
4      else if message.molecule does
       not exist then
5          reply with RESP_REMOVED;
6      else if message = FETCH then
7          clear molecule.list;
8          reply with molecule;
9      else if molecule has a
       commitment then
10         reply with RESP_TAKEN;
11     else if message = QUERY then
12         add sender to molecule.list;
13         reply with RESP_OK;
14     else if message =
       COMMITMENT then
15         SortRequesters(molecule);
16         if molecule.locker = sender
           then
17             reply with RESP_OK;
18         else
19             reply with
               RESP_TAKEN;
20 begin SortRequesters(molecule)
21     foreach pair of requesters in
       molecule.list do
22         if req_j.no_r < req_i.no_r
           then
23             put req_j before req_i;
24             continue;
25         if req_j.id < req_i.id then
26             put req_j before req_i;
27     molecule.locker ⇐ molecule.list
       (0);
```

Algorithm 3: Optimistic Sub-protocol — Requester.

```
1  on event combination found
2      foreach molecule in combination do
3          dispatch FETCH(molecule);
4  on event response received
5      if response ≠ RESP_MOLECULE then
6          Abandon(combination);
7          return;
8      add response.molecule to reaction_args;
9      if all responses have arrived then
10         NotifyHolders(combination);
11         Reaction(reaction_args);
12 begin NotifyHolders(combination)
13     foreach molecule in combination do
14         dispatch REACTION(molecule);
15 begin Abandon(combination)
16     foreach molecule in combination do
17         dispatch GIVE_UP(molecule);
```

Algorithm 4: Optimistic Sub-protocol — Holder.

```
1  on event message received
2      if message = GIVE_UP then
3          molecule.state ⇐ free;
4      else if message = REACTION then
5          remove molecule;
6      else if message.molecule does not exist then
7          reply with RESP_REMOVED;
8      else if molecule.state = taken then
9          reply with RESP_TAKEN;
10     else
11         molecule.state ⇐ taken;
12         reply with RESP_MOLECULE;
```

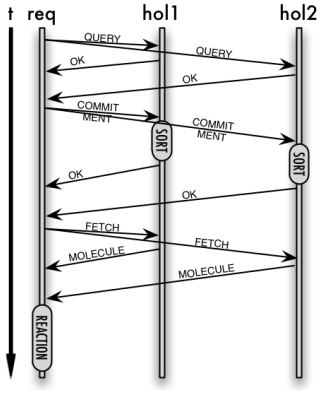

Figure 1: Pessimistic exchanges.

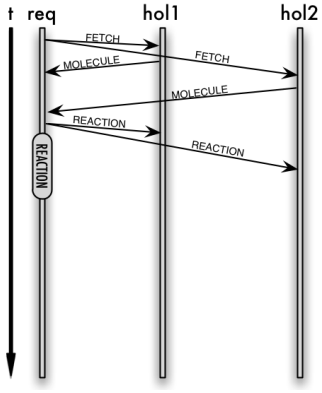

Figure 2: Optimistic exchanges.

3.3. Sub-protocol Mixing

During its execution, a program typically can pass through two different stages. The first one is the highly reactive stage, which is characterised by a high volume of possible concurrent reactions. In such a scenario, the use of the pessimistic sub-protocol would lead to superfluous network traffic, since the probability of a reaction's success is rather high. Thus, the optimistic approach is enough to deal with concurrent accesses to molecules. The second stage is the quiet stage, when there is a relatively small number of possible reactions. Since this entails highly probable conflicts between nodes, the pessimistic sub-protocol has to be employed in order to ensure the liveness of the system. Thus, the execution environment has to be able to adapt to changes and *switch* to the desired protocol accordingly. Moreover, these protocols have to be able to *coexist* in the same environment, as different nodes may act according to different modalities at the same time.

3.3.1. Switching

Ideally, the execution environment should be perceived as a whole in which the switch happens unanimously and simultaneously. Obviously, a global view of the reaction potential cannot be maintained in a large-scale system. Instead, each node independently decides which sub-protocol to employ for each reaction. The decision is first based on a node's local success rate, denoted σ_{local}, computed on the basis of the success history of the last queries the node issued. In order not to base the decision only on its local observations, a node also keeps track of local success rates of other nodes; each time a node receives a request or a reply message, the sender supplies it with its own current history-based success rate, stored into a list (of tunable size). Note that when there are multiple rules being executed, the node takes into account only the information relevant to the rule it is currently executing, as noted in Section 3.5. We denote σ the overall success rate, computed as the weighted arithmetic mean of a node's local success rate and the ones collected from other nodes. Finally, the decision as to which protocol to employ depends on the rule a node wishes to execute. More specifically, it is determined by the number of the rule's arguments, since the more molecules the rule needs, the harder it is to assure they will all be obtained. To grab r molecules, a node employs the optimistic sub-protocol if and only if $\sigma^r \geq s$, where r is the number of arguments the chosen rule has and s is a predefined threshold value. If the inequality is not satisfied, the node employs the pessimistic sub-protocol. Even though a more in-depth discussion about the value of the switch threshold falls out of the scope of this paper, we show its influence on the protocol's performance in Section 4.

3.3.2. Coexistence

Due to the locality of the switch between sub-protocols, not all participants in the system will perform it in the exact same moment, leading to possible inconsistencies in the system, where some nodes try to grab the same molecules using different sub-protocols. In order to distinguish between *optimistic* and *pessimistic* requests, each requester incorporates a *request type* field into messages.

Based on this field, the node holding the conflicting molecule gives priority to nodes employing the more conservative, pessimistic algorithm.

Although this decision discourages *optimistic* nodes and sets them back temporarily, it ensures that a node will be able to grab the molecules it needs *eventually*, since pessimism is favoured over optimism. When pessimistic nodes compete for molecules, one of them is surely going to perform its reaction as per the total order. On the other hand, when solely optimistic nodes compete for molecules, all of their reactions might be aborted since the optimistic sub-protocol does not guarantee the system's liveness. Consequently, pessimistic requests have a higher chance of being concluded by a reaction.

3.4. Dormant Nodes

During the quiet stage, the system might reach a point where $n \gg m$ (n denotes the number of nodes in the system, while m represents the number of molecules). In this extreme scenario, having more nodes represents a burden for the system, as most of the requests sent for molecules will ultimately be rejected, elevating the network traffic without speeding up the progression of the computation. Thus we introduce the notion of *dormant pessimistic nodes* — nodes which are using the pessimistic sub-protocol to capture molecules, but do so less often than usual. When a node switches to the pessimistic sub-protocol, it starts counting the number of consecutively aborted reactions. Once this number reaches a threshold a, it puts itself to *sleep* for a predefined amount of time δ — it becomes a *dormant* node. It then *wakes up* and tries to capture another combination of molecules. In case it succeeds, it becomes an active pessimistic node again. Otherwise, it returns to the dormant state for a δ amount of time, and so forth. In order to avoid *massive awakenings* of nodes, *i.e.* the simultaneous resumption of activities of a large number of dormant nodes, we allow the actual amount of time a node spends as dormant to vary by a constant ϵ_δ from δ: before putting itself to sleep, a node randomly chooses the number of steps it is going to sleep for from the interval $[\delta - \epsilon_\delta, \delta + \epsilon_\delta]$. Dormant nodes do not put their entire execution on hold — they are still active in the system as molecule holders. Note that a discussion about concrete values of δ, ϵ_δ and a and their fine-tuning is out of the scope of this paper.

3.5. Execution of Multiple Rules

Thus far we focused on the protocol and its various aspects carrying the assumption that there is only one rule in the program. However, in practice, a small fraction of chemical programs contain only one rule to be used in reactions, as one-rule programs appear rather limited when addressing more complex problems. As every node tries to carry out reactions, when multiple rules are present, each of them has to decide which of the rules it is going to employ in a given cycle. In the remainder, we assume the number of nodes executing the program is greater than the number of rules, *i.e.* $n > n_r$. We refer to the rule being executed by a node at a given moment as its *active rule*.

In order for the computation to be done as smoothly and efficiently as possible, certain constraints of the programming model have to be taken into account.

Firstly, a node's decision to switch from one sub-protocol to the other should take into account only the grabs it has tried to do for the rule it is using at the moment. Secondly, due to the paradigm's non-determinism and lack of sequentiality, a rule might be triggered at any given point in time: if a rule is used at time T, but not at time $T + \Delta T_1$, there is no guarantee that it will not be used again at time $T + \Delta T_1 + \Delta T_2$. Finally, the interdependency of rules influences the flow of the execution. Two rules can be: (i) *independent* (they can be concurrently executed), (ii) *dependent* (the product of one rule can be used as input by the other), and (iii) *circularly dependent* (the product of one rule can be used as input by the other and *vice versa*).

3.5.1. Multiple Success Rates

As noted earlier, while deciding which of the sub-protocols to employ to grab molecules, a node should take into account exclusively the capture attempts made while executing the currently active rule. Thus, the calculation of the success rate is adapted as follows. Now, each node manages a separate local success grab history list for each rule. Analogously, a separate list of observed remote successes is maintained per rule on each node. Consequently, a node is able to calculate multiple success rates (σ_i), one per rule i, and base its switch decision solely on information relevant to the active rule. Finally, the exchanged messages are expanded with one more field: the identifier of the rule for which the success rate is contained in the message. This way, nodes are able to differentiate success rates for distinct rules and place them in the correct lists. Note that, since a node bases its sub-protocol switch decision only for its currently active rule, while the threshold value is set globally for all rules, its interpretation depends on the rule for which the local switch decision is being taken.

3.5.2. Initial Rule Assignment

To ensure every rule is executed, n_r nodes are each permanently assigned a rule. These nodes are called *rule keepers* and they are selected based on the hash identifiers assigned to the rules: a node N is the rule keeper for a rule R if its node identifier is numerically the closest to the rule's hash identifier. Note that, in case a node, according to the hash function, should be the rule keeper for more than one rule, it may delegate the responsibility for all but one of them to other randomly-selected nodes. Rule keepers try to execute their assigned rules all throughout the computation — they behave as if only one rule (the one they execute) is present in the system. The rest of the nodes ($n - n_r$ of them) pick randomly one of the rules in the program with which to start the execution.

3.5.3. Changing the Active Rule

Even though rule keepers ensure the execution of every rule, the reaction potential of a rule varies throughout the computation; depending on the state of the program at a given moment, more reactants may be present for one rule than another. Thus, nodes ought to be able to change their active rules during

the execution based on the reaction potential of the rules. While a node is trying to obtain molecules using the optimistic sub-protocol, a change of the active rule is not being considered, as using this sub-protocol is an indicator of the active rule's high reaction potential. Hence, a change may occur when and only when a node is employing the pessimistic sub-protocol. Given the facts that a rule's reaction potential can be derived from a node's success rate and that every node keeps track of success rates for all of the rules, every node has got a good estimation of the reaction potential of each rule.

A node changes its active rule if the following conditions are met:

1. the node is currently using the pessimistic sub-protocol;
2. the node did not succeed to perform a reaction in the previous cycle;
3. the success rate for the active rule observed by the node is the smallest success rate when compared to all of the other rules' success rates.

If the above conditions are fulfilled, the node changes its active rule to the one with the highest success rate known. If the current σ value of the newly selected active rule permits the node to employ the optimistic protocol, it resets its grab history and sets its σ to 1, since it means that its reaction potential is high, entailing that a fair amount of reactions will be done switching to this rule. Otherwise, it means that the new active rule's reaction potential is also low. In this case the node will, for reasons described in Section 3.4, become a dormant node immediately after changing its active rule. Doing so, when it wakes up, it will start trying to apply the new active rule.

3.5.4. Discussion

The reader might have noticed that the presented rule-changing algorithm is a greedy one with respect to both time — *when* to change to another rule — and space — *which* rule to change to. Indeed, by adopting a policy of *late rule changing*, whereby a node changes rules only if its success rate is the worst it knows of, the subset of nodes executing a given rule consumes most of the rule's input molecules. The greediness with respect to space is manifested in the policy to switch to the rule with the highest success rate known to the node about to change rules. While switching to any rule with a value of σ higher than the active rule's would improve the execution, picking the highest one ensures the execution's *optimality*, in the sense that the node choosing it is guaranteed to encounter the least number of conflicts.

Combined, these two levels of greediness assure that (i) all of the reactions which can be done for a rule will be done, in the sense that if molecules capable of reactivating a rule appear, it is going to be chosen for execution; and that (ii) nodes avoid conflicts as much as possible. It should be pointed out that, due to the fact that there is no global view of the system as nodes build up their knowledge based on the communication with other nodes and that they choose a rule at random during initialisation, nodes will decide to execute different rules at distinct times. Moreover, they will not uniformly choose the same rule.

3.6. Proof of Correctness

To be *correct*, this protocol must guarantee two properties:

- *safety*: a molecule is used in at most one reaction (as we consider that every reaction consumes all of the molecules entering it);
- *liveness*: if a node sends a request infinitely often, it will eventually succeed in capturing the molecules, provided the requested molecules are still available.

3.6.1. Proof of Safety

Even though multiple *servers* (molecules holders) and multiple *clients* (molecule requesters) are involved in the process, safety is straightforward to prove, because both sub-protocols are synchronised by molecule requesters.

Theorem 1. *A molecule is consumed in at most one reaction.*

Proof. As visible in Algorithms 1 and 3, before performing reactions requesters wait for the responses of all concerned molecule holders. A reaction is carried out if and only if all of the molecules are present on the requester. Does it not receive a molecule, a requester renounces performing the reaction by executing the `Abandon` routine, giving back all of the molecules it has captured. Additionally, when employing the pessimistic sub-protocol, a requester has to pass through three synchronisation barriers, one after each phase.

On the other side, each holder locally enforces conflict resolution. Because both sub-protocols have conflict resolution policies which ensure that a molecule can be given to only one requester, a molecule will be consumed in at most one reaction. This is observable in Algorithms 2 and 4, where a molecule's state changes based on the arrived request. Once it has been promised or given to a requester, others receive either a *RESP_TAKEN* or a *RESP_REMOVED* message even if the molecule can be given back later to the holder due to a requester's call to the `Abandon` routine.

Finally, there are two cases of conflict between the two protocols. When an optimistic request arrives before a pessimistic one, the pessimistic request is aborted because the molecule has already been reserved by the optimistic requester. On the other hand, if a pessimistic request arrives first, the optimistic request is aborted in favour of the pessimistic one. □

3.6.2. Liveness Proof

To prove the liveness property, we show that:

- the protocol is deadlock-free;
- if no successful reaction happens in the system, nodes eventually switch to the pessimistic protocol;
- if several pessimistic requesters are in conflict, at least one reaction is not aborted;

In addition, we show that a node cannot see its reactions infinitely aborted, *i.e.* that the protocol is starvation-free.

Lemma 1. *A node's execution cannot be blocked infinitely.*

Proof. Although requesters compete against each other for molecules, ultimately the decision is taken unilaterally by the holders on which the molecules reside. This decision is communicated as a response to each request. Due to the usage of reliable FIFO channels, a requester will always get a response for each sent request. Based on the received responses, it will either perform the reaction or abort it and continue its execution. □

Lemma 2. *If an optimistic node sees its reactions continuously aborted, it eventually switches to the pessimistic sub-protocol.*

Proof. When a request of a node is aborted, the node decreases its value of σ (see Section 3.3). With each message sent, a node includes the information about its local σ, and collects the values received from other nodes. If there are many conflicts during a certain period of time, all the more so if there is no successful reaction, the local values of σ of all of the nodes decrease. This effect leads to a situation where the computed value of σ^r for all new reactions is lower than the threshold s, which forces nodes to use the pessimistic protocol upon the initiation of new requests. □

Lemma 3. *Eventually, at least one node will succeed in performing a reaction.*

Proof. Initially, and hopefully most of the time, nodes use the optimistic sub-protocol for their requests. In case of a conflict between two optimistic requesters, both requests can easily be aborted. Consider the example where two concurrent requesters try to capture two molecules, A and B. If the first requester succeeds in grabbing A while the second captures B, then the two requests will be aborted. If such scenarios persist, as per Lemma 2, nodes will switch to the pessimistic sub-protocol.

For the pessimistic sub-protocol, we define a total order based on the number of successfully completed reactions by a node and its unique id. In case of a conflict, all of the reactions might be aborted except for one — the reaction initiated by the node which comes first as per the total order. Since that node has got the highest priority system-wide, all of the holders it contacts will decide in its favour. □

Following Lemmas 1—3 we have:

Theorem 2. *The protocol assures the system's liveness property holds.*

We now prove that the protocol is starvation-free.

Lemma 4. *A node cannot see its reactions infinitely aborted.*

Proof. There are two possible outcome scenarios when a node enters in a conflict over molecules: (i) the molecules exist long enough for a node to capture them; and (ii) the molecules are taken by another node. Following Lemma 2, a node trying to obtain molecules will eventually switch to the pessimistic sub-protocol.

Because the total order is based on the number of successful reactions, if the node, in case of an abort, tries again infinitely to request molecules for its reaction, eventually, provided the requested molecules are still available, the reaction will take place, given the fact that its position moves up the total order when other nodes succeed in executing their reactions.

If, however, the node does not have the highest priority amongst the nodes in conflict for the molecules, another node will grab them, in this way raising other nodes' positions up the total order. The original node will then try to grab another combination of molecules. It will change the combination until it becomes the node with the least number of reactions performed, at which point it will have the highest priority in the total order. □

3.6.3. Convergence Time

When presenting algorithms for atomic capture, it is common to study their convergence times. However, any discussion about convergence when dealing with the chemical programming model is not feasible, as convergence itself, and thus the convergence time, is an application-specific property. However, the next section presents an evaluation of the proposed algorithm, and sheds some light on the subject.

4. Evaluation

Our protocol was simulated in order to better capture its performance. We developed a Python-based, discrete-time simulator, including a DHT layer performing the random dissemination of a set of molecules over the nodes, on top of which the layer containing the capture protocol itself was built. At this layer, any message issued at step t will be received and processed by the destination node at time $t + 1$. Moreover, each time a capture attempt either led to a reaction, or to an abort, the node tries to fetch another set of r randomly chosen molecules, where r depends on the program being simulated atop the protocol.

Unless otherwise noted, all presented experiments simulate a system of 250 nodes trying to execute a chemical program containing a solution with 15000 molecules. The reactions' durations are assumed negligible, as this allows us to concentrate exclusively on evaluating the capture protocol itself, without having to deal with application-specific problems. For all of the simulations, we used the following constants: $a = 10$; $\delta = 20$; $\epsilon_\delta = 4$; and $s = 0.7$. Each simulation was run 50 times and the figures presented below show the values obtained by averaging result data from these runs. As the deviation for each simulation is negligible, we here present only the averaged values.

There are two sets of experiments. In the first one we extensively tested the protocol's behaviour, its performance and the network traffic generated by a

simple program with a single rule. The second set of experiments examines the system's behaviour when faced with the execution of programs with multiple rules.

For the purpose of evaluating the protocol and its characteristics we simulated five different programs on top of the protocol itself. Note that the programs do not represent concrete implementations of applications since the actual reactions and their results are not taken into account. Rather, they were conceived in such a way as to examine the protocol in detail. The programs are designed to ensure inertia will be reached in a finite number of steps, since the problem of distributed inertia detection is out of the scope of this paper. These five programs cover all of the rule-dependency patterns described in Section 3.5, and thus provide a complete insight into the protocol's characteristics. The next section presents the results obtained by simulating the single-rule program, while Section 4.2 describes the multiple-rule programs and the outcomes of their experimentations.

4.1. Experiments with One Rule

In the first set of experiments we concentrated on the characteristics of the devised protocol itself, namely its performance and network overhead. Thus, the single-rule program was used throughout this set of experiments.

Single-rule Program. The first program simulated is a simple one consisting only of a straightforward rule which simply consumes two molecules without producing new ones. Having only one rule in the solution allows us to concentrate and analyse solely the protocol, its sub-protocols and the switch between them.

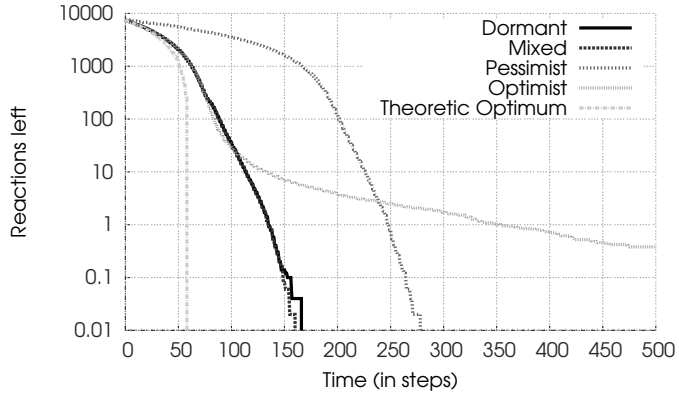

Figure 3: Performance comparison of the protocol's variants.

Experiment 1 (Execution Time). Firstly we evaluate separately the performance characteristics of both sub-protocols. Figure 3 shows the averaged number of

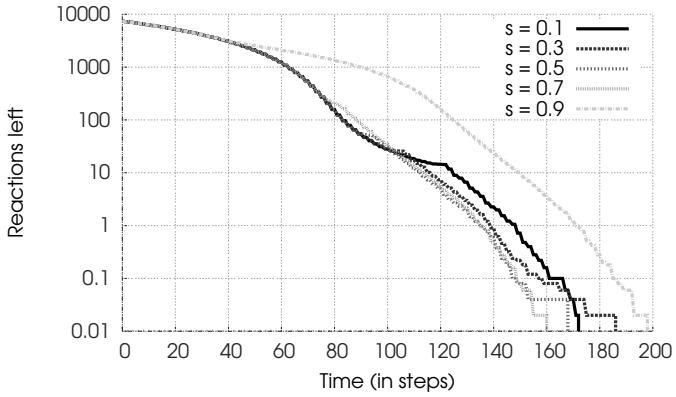

Figure 4: Execution time for different switch thresholds.

reactions left to execute at each step, until inertia, using only the optimistic mode, only the pessimistic mode, and the complete protocol with switches between protocols (using $s = 0.7$) with and without the optimisation of dormant nodes, respectively. The *theoretic optimum* curve represents the amount of steps needed to complete the execution in an *ideal* distributed system, *i.e.* with the highest possible parallelism degree and no conflicts during the whole computation. Considering that we need at least two steps to fetch molecules (one to request them and one to receive them), this hypothetical ideal system needs $2 * \frac{m}{nr}$ steps to conclude the computation. This represents a lower bound on the number of steps, regardless of the model of computation, be it chemical or other. Note that a logarithmic scale is used for the number of reactions left. The figure shows that, when using only the *optimistic* protocol, there is a strong decline in the number of reactions left at the beginning of the computation, *i.e.* when a lot of reactions are possible and that thus there are only few conflicts in the requests. However, it gets harder for nodes to grab molecules when this number declines. In fact, the system is not even able, for most of the runs, to finish the execution, as the few reactions left are never executed, constantly generating conflicts at fetch time. When the nodes are all *pessimistic*, there is a steady, linear decrease in the number of reactions left, and the system is able to reach inertia in a reasonable amount of time, thanks to the liveness ensured in this mode. For most steps, the *mixed* curve traces the exact same path as the *optimistic* one, which means that during this period the nodes employ the optimistic sub-protocol. However, at the end, the system is able to quickly finish the execution as an aftermath of switching to the pessimistic sub-protocol. After the switch, it diverges from the optimistic one to mimic the pessimistic curve, exhibiting a 42% performance boost compared to the performance of the pessimistic sub-protocol. Comparing the theoretic optimum to our protocol, we notice an increase of 166% in the number of steps needed to reach inertia. This is understandable, because there is usually a coordinator in centralised systems

with which conflict situations can be circumvented, but it opens the door to serious defaults, such as single-point-of-failure or bottleneck problems. Finally, as far as performance is concerned, including dormant nodes leads to similar results.

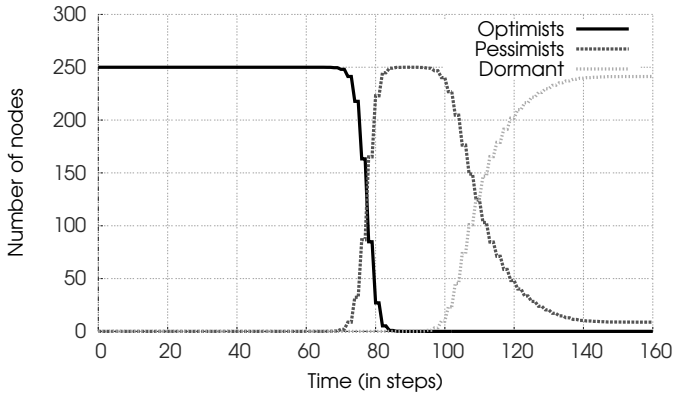

Figure 5: Nodes employing optimistic and pessimistic sub-protocols and dormant nodes per step.

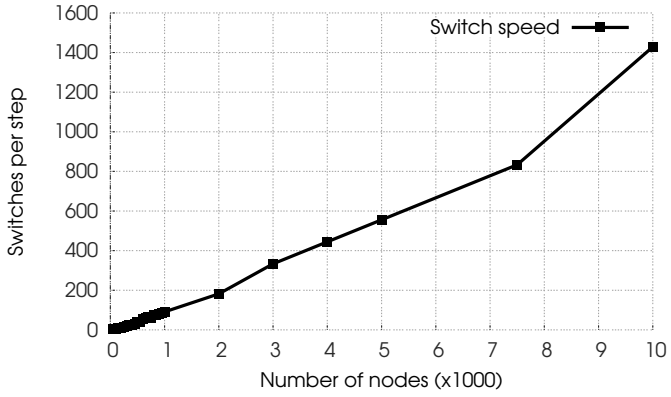

Figure 6: Switch speed expressed as number of switches per step (for a constant number of molecules).

Experiment 2 (Switch Threshold Impact). Next, we want to assess the impact of s (the switch threshold) on the overall performance of the system. Figure 4 depicts, in the same logarithmic scale, the number of reactions left after each step for different threshold values, varying from 0.1 to 0.9. As suspected, the curves overlap during most steps, most nodes employing the optimistic sub-protocol. The first curve to diverge is the one where the switch threshold is set very high, to $s = 0.9$. Because the system depicted by that curve did not

fully exploit the optimistic sub-protocol, it is the last to finish the execution. Although slightly, the other curves start diverging at different moments, and, thus, complete the execution after a different number of steps. Figure 4 shows that, out of the five values tested for the switch threshold, $s = 0.7$ yields the best performance results in this particular scenario. Looking at completion times for different switch threshold values brings us to the conclusion that the switch threshold can have a significant impact on performance; in this case the execution time can be decreased by up to 20%. Finding an optimal value for s for any application falls out of the scope of this paper.

Experiment 3 (Switch Behaviour). Here we examine the properties of the process of switching from one protocol to the other, represented in Figures 5 and 6. Figure 5 depicts the evolution of the number of nodes in each mode during the execution. We can see that, at the beginning of the execution, all of the nodes start grabbing molecules by using the optimistic sub-protocol. The switch happens about half way through the execution. Around that time, *optimistic* nodes start aborting more and more reactions, and thus can no longer efficiently capture molecules, so they switch to the pessimistic sub-protocol. We observe that, thanks to the systematic exchanges of local σ values, nodes in the system reach a global consensus rather quickly — for a system with 250 nodes, at most 15 steps are needed for all of the nodes to switch to the pessimistic protocol. In other words, the complete transition from using the optimistic sub-protocol to using the pessimistic one constitutes at most 10% of the execution time. For the following 15 steps all of the nodes are active – and pessimistic – and try to capture molecules. However, as the concentration of molecules further drops, the number of dormant nodes increases, in this way reducing network traffic and allowing the still active nodes to capture the wanted molecules with more ease. Note that, while the overall number of dormant nodes increases, nodes wake up after a certain period and become pessimistic again. Still, one can observe that, overall, there are more nodes asleep than pessimistic ones.

Figure 6 illustrates the number of nodes that switch from the optimistic to the pessimistic sub-protocol on each step during the transition period. One can observe that the more nodes in the system, the greater the number of nodes that switch per step. This behaviour comes from the fact that an increase in the number of nodes implies a greater accuracy of the system's state estimation σ as each node communicates with a wider spectre of nodes. This shows that the system, regardless of its size, can react quickly to changes, even though there is no global view of the situation.

As discussed in Section 3.3, pessimistic requests are favoured over optimistic ones. However, we also conducted simulations when the inverse is true, *i.e.* when optimistic requests are favoured. The results obtained are similar to those shown in Figure 5: there is no difference in the total execution time nor in the switch speed. The figure is, therefore, omitted from the paper. The similarities stem from *(i)* the quick propagation of local σ values; and *(ii)* the pessimistic requests' higher chance of completing a capture cycle. Indeed, even though optimistic requests are favoured, as the concentration of available molecules in

the system drops, it get harder for nodes employing the optimistic sub-protocol to capture all of the molecules needed for their reactions.

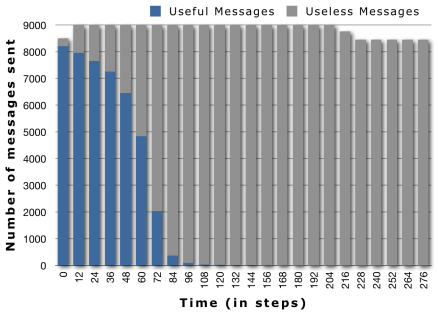

Figure 7: Generated messages when only the optimistic sub-protocol is active.

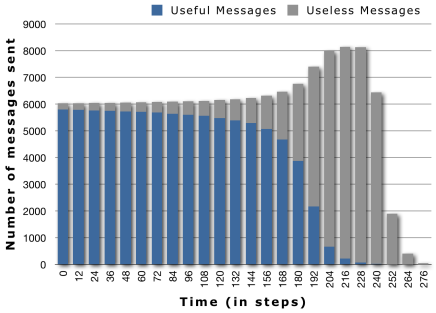

Figure 8: Generated messages when only the pessimistic sub-protocol is active.

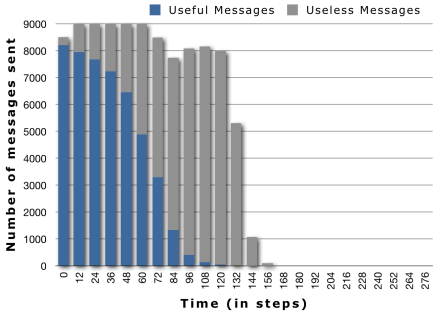

Figure 9: Messages generated by the proposed protocol (without the dormant state).

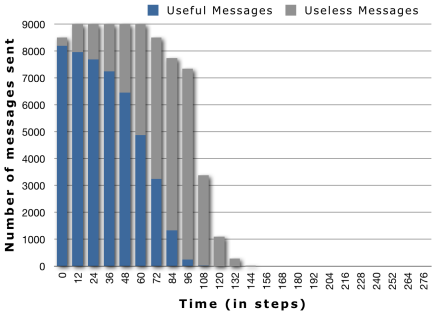

Figure 10: Messages generated by the proposed protocol when pessimistic nodes can become dormant.

Experiment 4 (Communication Costs). Next, we investigate the communication costs involved in the process. Figures 7, 8, 9 and 10 depict the number of messages sent per cycle in a system of 250 nodes. One cycle comprises 12 simulation steps, as it is the lowest common multiple of 4 and 6; at most 4 steps are needed for the optimistic sub-protocol to complete, and at most 6 steps for the pessimistic sub-protocol. The messages are classified into two categories: *useful* messages (ones which led to a reaction, in blue) and *useless* messages (those which did not induce a reaction, in grey). When looking at the communication costs of the optimistic sub-protocol (Figure 7), one can observe that a high volume of reactions is done in the beginning of the execution with a small percentage of conflicts, and thus a small amount of useless messages. However, as the execution progresses, the percentage of useful messages drops rapidly, while the total number of messages is kept high. Figure 8 shows that the pessimistic sub-protocol consumes less messages, with the percentage of useful

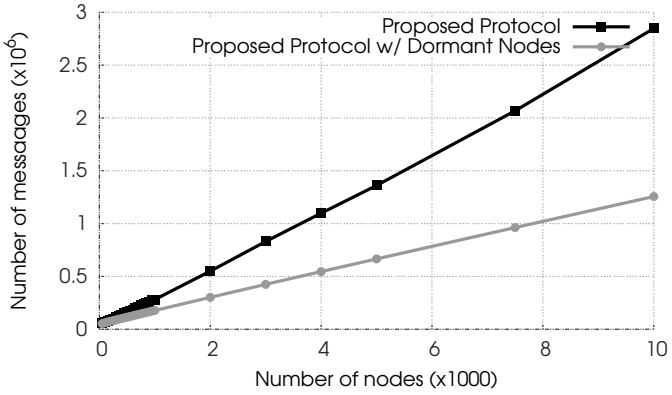

Figure 11: Total number of messages sent when varying the number of nodes in the system.

messages dropping steadily and slowly. At the same time, as there are less and less molecules in the system, the total number of messages slowly grows before peaking at 8000 messages per cycle and then rapidly decreasing towards the end of the execution. When comparing Figure 9 to the previous two, we note that the protocol takes over the best properties of both of its sub-protocols. Firstly, it takes over the elevated number of useful messages of the optimistic sub-protocol. After the switch, the pessimistic protocol kicks in, bringing with it a decrease in the total number of messages. Consequently, using a simplistic and lightweight sub-protocol when possible and then falling back on a heavier one reduces network traffic and improves scalability — a decrease of 30% in the number of messages can be observed when compared to the pessimistic sub-protocol alone. In addition, Figure 10 reveals that using the policy of dormant nodes further improves the scalability of the protocol, as it significantly reduces the total number of messages towards the end of the execution where there is the highest number of conflicts. Finally, Figure 11 shows the number of messages sent when the system's size varies from 50 to up to 10000 nodes and confirms the protocol's scalability: the number of messages linearly grows with the system's size. Moreover, the scalability greatly improves by using dormant nodes — the slope is gentler, rapidly widening the gap between the two curves.

4.2. Experiments Involving Multiple Rules

Now we are shifting our focus onto the rule-changing mechanism described in Section 3.5. We want to examine its decision-making policy as well as look at the behaviour of the protocol during the execution of programs comprised of multiple rules. There are four experiments in this set, each evaluating one of four multiple-rule programs, the descriptions of which follow.

4.2.1. Multiple-rule Test Programs

Independent-rules Program. A natural extension of the single-rule program, this one contains three rules — $R0$, $R1$, $R2$ — which are *independent*, *i.e.* no two

rules consume or produce the same type of molecules. Thus, reactions using these rules can be done fully concurrently, without any interference, mutual exclusion or synchronisation. The rules consume two molecules of type $T0$, $T1$ and $T2$, respectively. They produce no output.

Figure 12: Input/output links between rules for the dependent-rules program.

Figure 13: Input/output links between rules for the circular program.

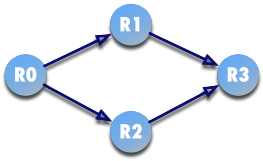

Figure 14: Input/output links between rules for the workflow program.

Dependent-rules Program. Developing the previous program further we come to the next one. In this new program, the three rules are now *dependent* — the molecules produced by one rule are consumed by another. These input/output links are illustrated in Figure 12. Each rule still consumes two molecules of its own type. However, to create the dependencies between them, in this program $R0$ produces two molecules of type $T1$ (used as input by $R1$), while $R1$ produces two molecules of type $T2$ (consumed by $R2$). $R2$ produces no output.

Circular Program. This program exploits the circular-dependency pattern, as shown on Figure 13. Its characteristics are the same as those of the dependent-rules program, except that another input/output link has been established between the rules $R2$ and $R0$ in order to create the circular flow diagram: $R2$ now produces a single molecule of type $T0$. Note that the fact that $R2$ produces less molecules (only one) than it consumes (two) ensures inertia will be reached; outputting two would cause an infinite execution loop.

Workflow Program. The last program is somewhat more complex than the others, as it is a small *split/merge workflow* of rules comprising both dependent and independent rules. The links between rules are depicted in Figure 14. It consists of four rules: $R0$, $R1$, $R2$ and $R3$. The rule $R0$ consumes two molecules of type $T0$ and produces two molecules: one of type $T1$, the other of type $T2$. These are used as input by $R1$ and $R2$, respectively. These rules can, thus, be run concurrently and independently of each other. $R1$ produces one molecule of type $T3$, while $R2$ produces one of type $T4$. Finally, their outputs are *merged* by the rule $R3$, which consumes one molecule per type — $T3$ and $T4$ — and produces no output.

4.2.2. Evaluation

Experiment 5 (Independent-rules Program). The first program containing multiple rules we examined was the simplest one — independent-rules. Figure 15 depicts the flow of its execution: the number of nodes executing each rule is

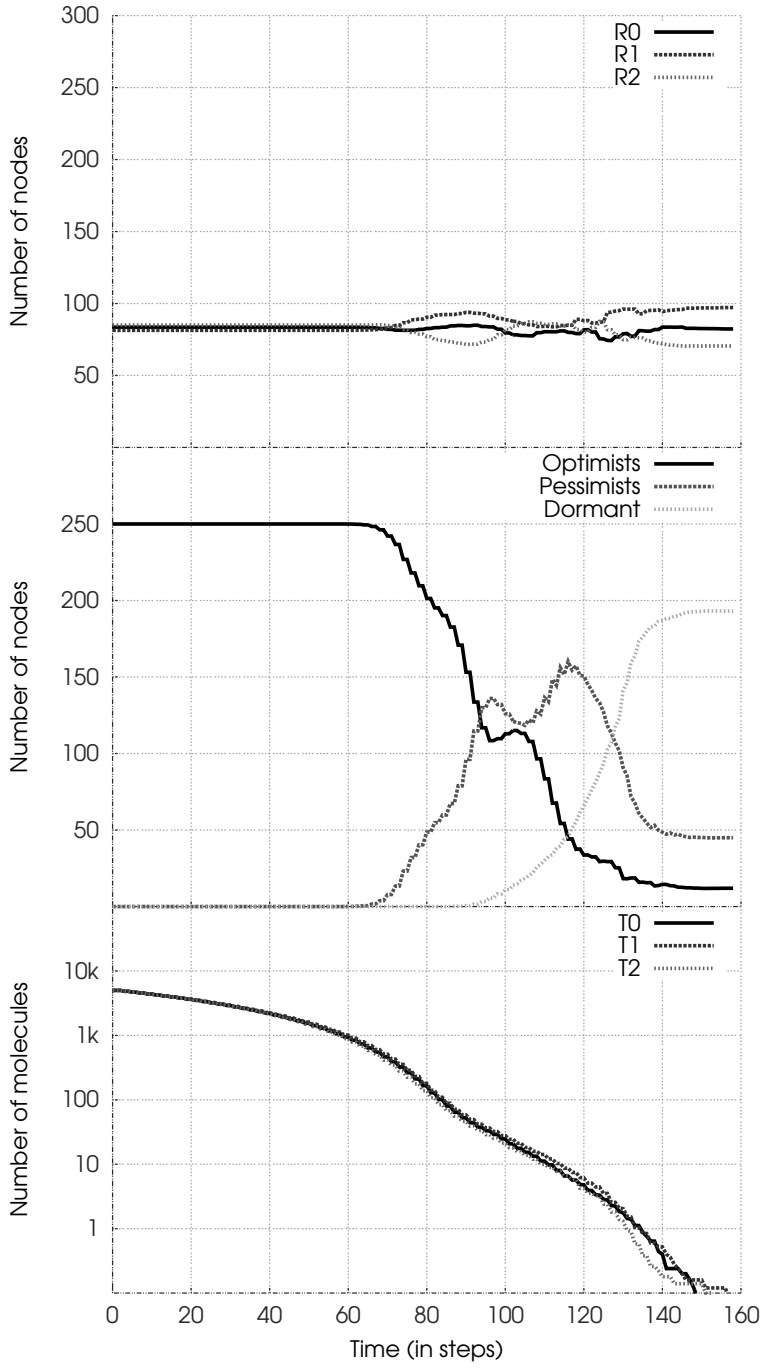

Figure 15: Number of nodes executing each rule (top), the number of pessimistic *vs* optimistic nodes (middle) and the number of molecules of each type in the solution (bottom, in logarithmic scale) during the execution of the independent program.

shown in the top of the figure, in the middle the number of nodes employing each sub-protocol is depicted, while the bottom illustrates, in logarithmic scale, the number of molecules of each type present in the solution. It reveals that an equal number of nodes execute each rule, which is to be expected since all of the rules are executable in parallel and are not in conflict (no two rules share their input molecule types). All the while, all of the nodes employ the optimistic sub-protocol as the concentration of molecules is high enough for nodes to avoid conflicts. As soon the number of optimists starts to decline (around step 75), the nodes start to change rules, causing the fluctuations observed in the upper part of the figure. From that point on we can see a constant decrease in the number of optimists since more and more nodes enter into conflict over molecules, which start to become rarer and rarer. At the same time, there is an increase in the number of pessimistic (and then dormant) nodes, suggesting that most nodes keep employing the pessimistic sub-protocol even after changing rules.

Due to the almost-perfectly equal distribution of nodes over rules they use for reactions we conclude that the rule-changing mechanism correctly decides which rule a node should execute. Moreover, as a result of changing rules, there are nodes employing the optimistic sub-protocol all throughout the execution, in this way speeding up the computation.

Experiment 6 (Dependent-rules Program). Figure 16 illustrates the course of the execution of the dependent-rules program. At the beginning of the execution, all of the nodes but $R1$'s and $R2$'s rule keepers, are applying the rule $R0$ since there are only molecules of type $T0$ present in the system. Indeed, the discovery protocol (abstracted out in this paper) is not able to discover molecules of other types, prompting the nodes to change their active rule to $R0$ immediately. Then, as the computation progresses, it becomes harder for nodes to grab $T0$-molecules and they start turning pessimistic. As $T1$-molecules appear, some nodes opt to change their active rule. More specifically, as there are more $T1$-molecules around step 75, most of the nodes choose to execute the rule $R1$, while a minority changes for $R2$ since the rule keeper of $R1$ managed to produce a few $T2$-molecules. Due to this change of rules, all of the nodes become optimistic again. Then, as they successfully perform reactions, the number of $T1$-molecules rapidly drops, inducing another cycle of *mass rule changing*. This time, $R2$ is favoured due to the high concentration of $T2$-molecules. This change prompts nodes to become optimistic again. Because there are some $T1$-molecules left around step 250, half of the nodes change back to $R1$ to complete its execution, causing a sudden drop in the number of optimists and the oscillation between being optimistic and pessimistic. At the same time, the number of dormant nodes increases, meaning that nodes increasingly perceive both rules as pessimistic. This is in accordance with the state of the solution — there are very few molecules left in the system. In spite of the pessimism, towards the end of the execution all of the nodes gradually switch back to $R2$, finishing the execution either as pessimists or dormant nodes.

The conclusion drawn from the experiment is that the local decisions taken by the rule-changing mechanism follow the flow of dependency between rules and

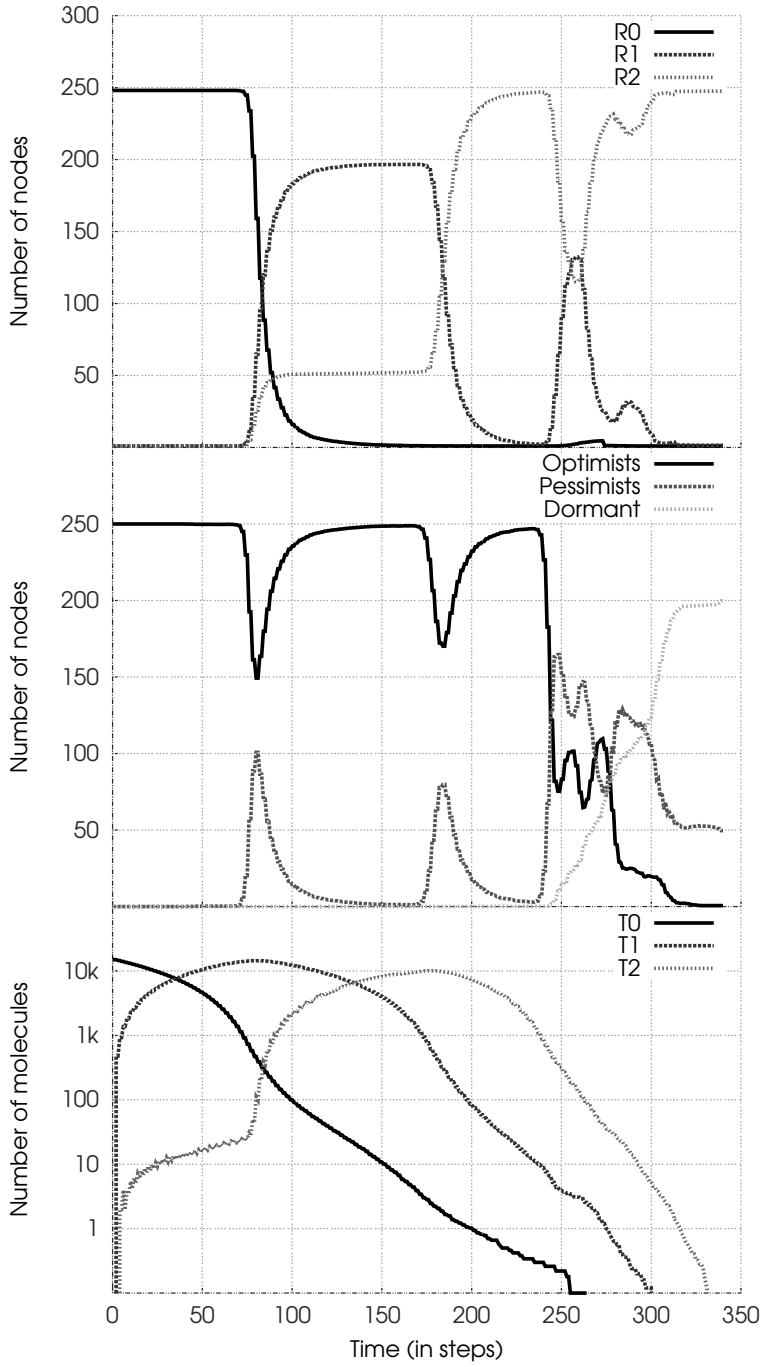

Figure 16: Number of nodes executing each rule (top), number of pessimistic *vs* optimistic nodes (middle) and number of molecules of each type in the solution (bottom, in logarithmic scale) during the execution of the dependent program.

are, thus, correctly taken in the case of multiple-rule dependency. In addition, changing rules causes the nodes to become optimistic again, which allows the system to progress faster.

Experiment 7 (Circular Program). Next, we simulated the circular program using two distinct initial configurations, for which the results are shown in Figures 17 and 18. In the first case the initial solution contained exclusively $T0$-molecules, while in the second all three types of molecules were equally represented.

When examining Figure 17, one can see that at the beginning of the execution all of the nodes use the rule $R0$, which is consistent with the fact that there are only $T0$-molecules in the solution. However, the concentration of $T1$-molecules rapidly grows, causing the nodes to pass to the execution of $R1$ once they have become pessimistic. In the same vein, around step 150 they opt for $R2$, after which we can see the $R0$-$R1$-$R2$ execution pattern appear again. As about step 200 the overall concentration of molecules is rather low, dormant nodes begin to appear, while the number of optimists quickly drops to zero. By the end of the execution, there are only a few molecules left and most of the nodes are thus dormant, while only a small minority completes the few remaining reactions.

The scenery drastically changes when all types of molecules are present in the initial solution, as shown on Figure 18. The course of the execution bears a strong resemblance to that of the independent-rules program (Figure 15): since all types of molecules are constantly being consumed and produced, the nodes are able to behave as if there are no dependencies between the rules. There are slight differences in the two programs, though. Unlike in the independent-rules program, here one can notice the cyclic change of the rules' dominance from Figure 17 (on a smaller scale though). Furthermore, the number of optimists drops much faster here towards the end of the execution because a decrease in concentration of molecules of one type implies an immediate decrease in concentration of all the others'.

This experiment shows that, while the dependency between rules plays an important role in the course of the execution, so do the data contained in the initial solution when it comes to cyclic dependences between rules. However, both the algorithm and the rule-changing mechanism are able to properly detect the reaction potential of rules in each case, and thus follow the dependency flow brought about by the program.

Experiment 8 (Workflow Program). In the last experiment, we observed the behaviour of the system during the execution of the workflow program. The results, depicted in Figure 19, show a substantial similarity to those of the dependent-rules program (Figure 16). This behaviour is to be expected, since this program is a variant of the dependent-rules program, whereby instead of one middle rule there are two parallel ones independent of each other.

The rule-changing pattern reveals that the nodes first massively execute $R0$ until the concentration of $T0$-molecules drops below those of $T1$- and $T2$-

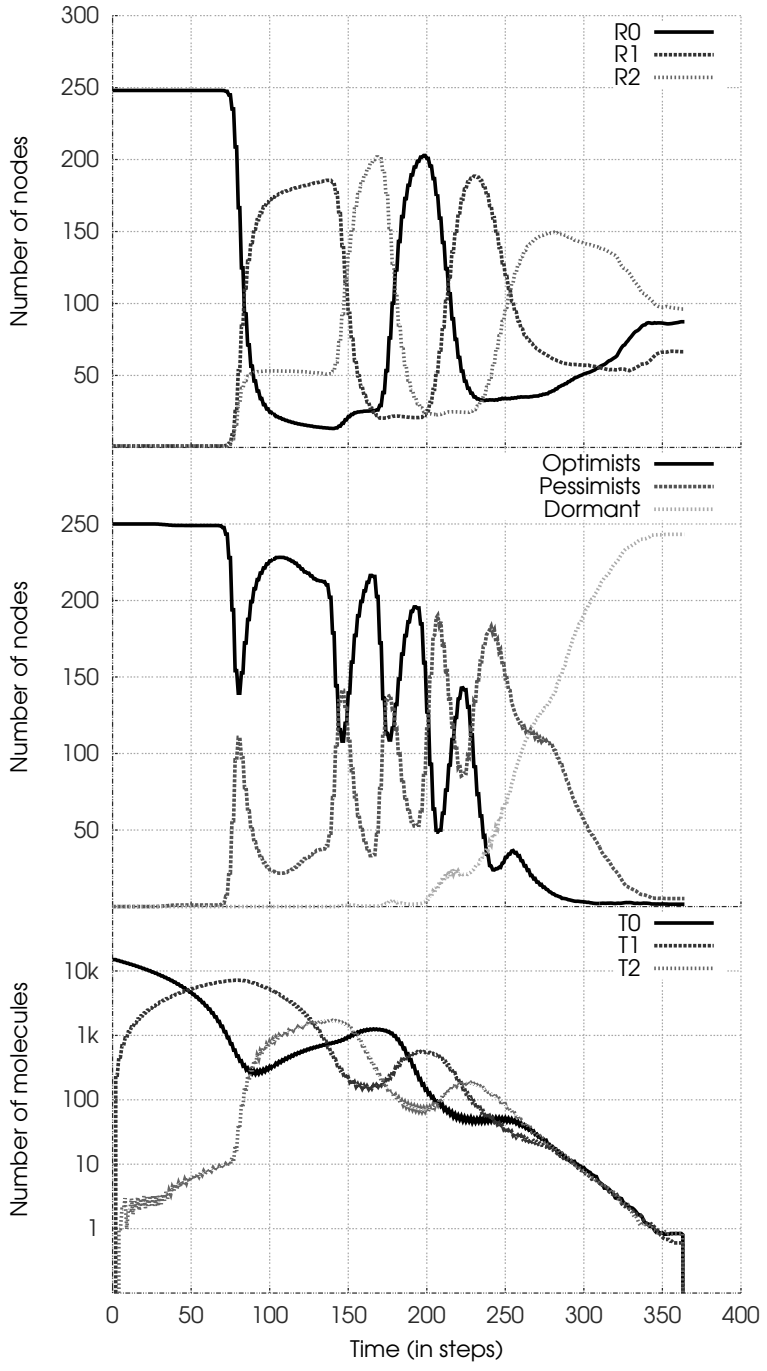

Figure 17: Number of nodes executing each rule (top), number of pessimistic *vs* optimistic nodes (middle) and number of molecules of each type in the solution (bottom, in logarithmic scale) during the execution of the circular program.

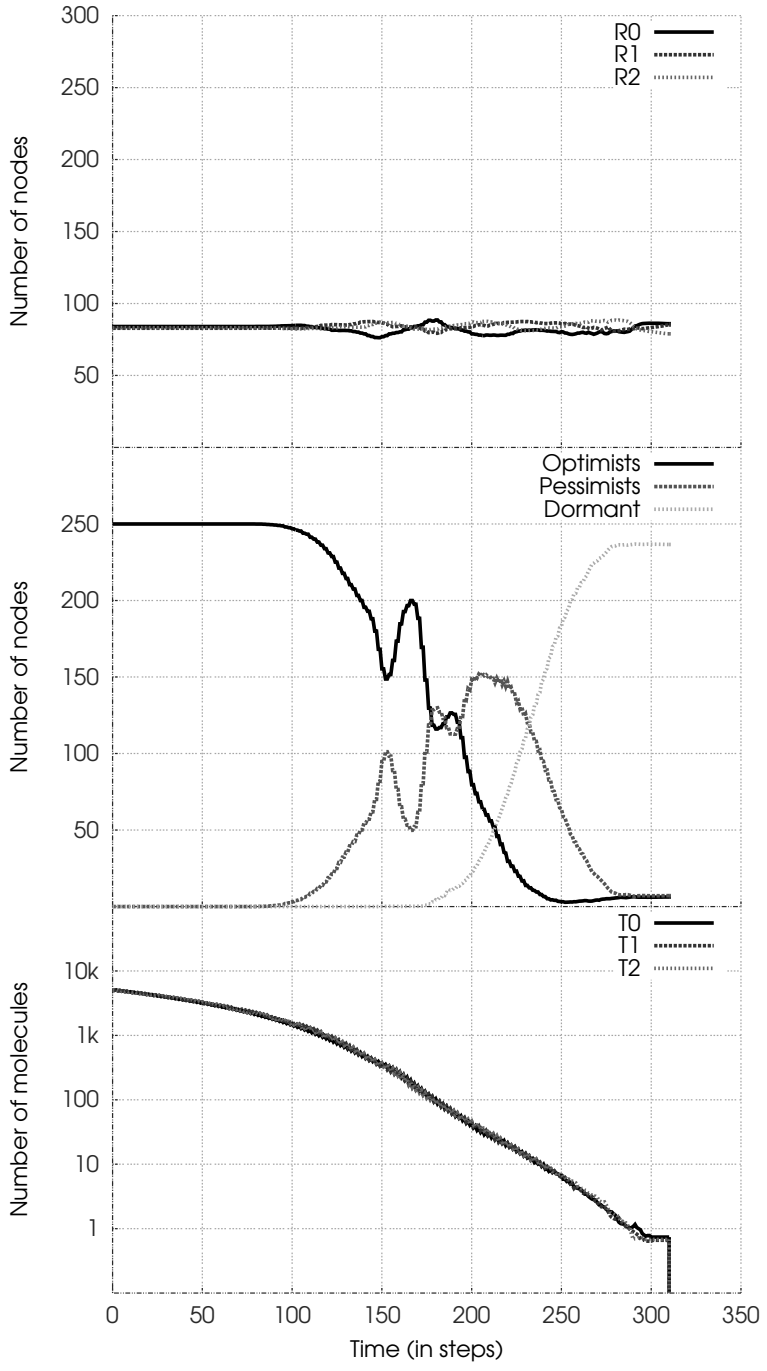

Figure 18: Number of nodes executing each rule (top), number of pessimistic *vs* optimistic nodes (middle) and number of molecules of each type in the solution (bottom, in logarithmic scale) during the execution of the circular program with different initial conditions.

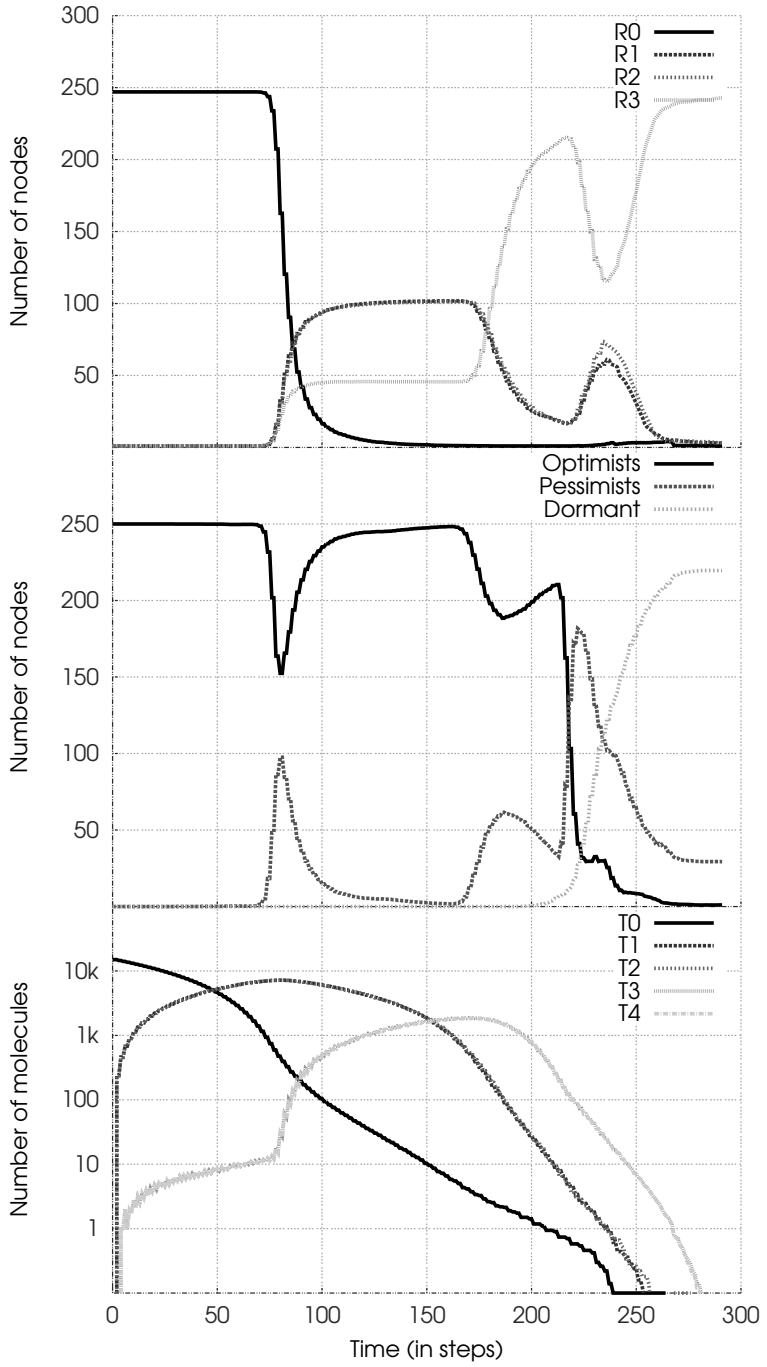

Figure 19: Number of nodes executing each rule (top) and the number of pessimistic *vs* optimistic nodes (middle) and the number of molecules of each type in the solution (bottom, in logarithmic scale) during the execution of the workflow program.

molecules. The nodes then distribute themselves over the rules in such a way as to consume mostly these molecules — around 100 nodes per rule for $R1$ and $R2$. As they produce $T3$- and $T4$-molecules, about 50 nodes start executing the last rule, $R3$. As $T1$- and $T2$-molecules are consumed at a faster pace, nodes start abandoning $R1$ and $R2$ and pick $R3$. All the while, most of the nodes are employing the optimistic sub-protocol. Then, in between steps 200 and 250 about half of the nodes change back to $R1$ and $R2$ in order to consume the remaining $T1$- and $T2$-molecules, respectively, causing a decrease in the number of optimists, due to the globally low concentration of molecules. Aat the end of the execution almost all of the nodes use $R3$, with most of them dormant. Indeed, as there are only a few reactions left at that point, it is impossible for most of the nodes to perform reactions in spite the fact that they have correctly picked the rule to execute.

This experiment confirms that the rule-changing mechanism is able to follow more complex dependency patterns. Moreover, we can see that during most of the execution the majority of nodes uses the optimistic sub-protocol, confirming that the protocol is able to adapt itself to the current situation in the system.

5. Related Works

The chemical paradigm was originally conceived for programs which need to be executed on parallel machines. The pioneering work of Banâtre *et al.* [3] provides two conceptual approaches to the implementation problem, in both of which each processor of a parallel machine holds a molecule and compares it with the molecules of all the other processors. A slightly different approach was proposed in the work of Linpeng *et al.* [14], where a program is executed by placing molecules on a strip, and then folding them over after each vertical comparison. Recently, Lin *et al.* developed a parser of GAMMA programs for their execution on a cluster exploiting GPU computing power [17]. All works mentioned exhibit significant speed-up properties, but the platforms on which the authors experimented were rather restricted.

Mutual exclusion and resource allocation algorithms have been studied extensively. Most research focuses on sharing one specific resource, or critical section, amongst many processes [27, 8]. A basic solution for the *k-out of-M* problem was given by Raynal [25]. This early work is a static permission-based algorithm in which only the number of a predefined set of resources varies from node to node. In addition, the solution supposes a global knowledge of the system. On the other hand, an execution environment for chemical programs is a dynamic system in which nodes need to obtain different molecules, which can be thought of as resources, at different times.

The three-phase commit protocol was originally proposed as a crash recovery protocol for distributed database systems [30]. The authors study the two-phase protocol and add to it a third, the so-called *prepare commit* phase, thanks to which they are able to obtain a system which is able to abort database transactions in any moment. Although in its essence similar to the three-phase commit protocol, the goal of the pessimistic sub-protocol proposed in this paper

is to secure the liveness of the system by ensuring that at least one node will be able to complete its reaction in a situation where multiple requesters are in conflict over different molecules.

6. Conclusion

While the chemical metaphor is gaining interest in the context of autonomic computing, the actual deployment of programs following the chemical programming model over distributed platforms is a widely open problem.

In this paper, we have described a protocol to capture several molecules atomically in an evolving multiset of objects distributed on top of a large-scale platform. The protocol consists in the association of two sub-protocols intended to face different levels of density of potential reactions in the multiset. By dynamically switching from one sub-protocol to the other, our protocol fully exploits their good properties (the low communication overhead and speed of the optimistic protocol, when the density of reactants is high, and the liveness guarantee of the pessimistic protocol, when this density drops), without suffering from their drawbacks. We also propose a communication-reduction scheme which is activated during the low-density period. Furthermore, we provide a rule-changing mechanism able to guide the nodes' computation when a program with multiple rules is being executed. The paper provides a formal proof of the protocol's correctness and its different aspects have been experimented with through simulation.

This protocol is part of an ambitious work which aims at building a distributed runtime for chemical programs. This work is also worth pursuing in that it revisits classical problems in distributed systems, but with the specificities of the chemical model in mind. In this way, this paper tackles the mutual exclusion with the liveness property as a system property while, while more traditionally, liveness is a process' property.

Among the directions planned for this work, one is to refine the execution model to, for instance, balance the load of reactions among the nodes of the platform.

References

[1] S. Abiteboul, M. Bienvenu, A. Galland, E. Antoine, A Rule-based Language for Web Data Management, in: Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of database systems (PODS'11), Athens, Greece, pp. 293–304.

[2] Ö. Babaoglu, G. Canright, A. Deutsch, G.D. Caro, F. Ducatelle, L.M. Gambardella, N. Ganguly, M. Jelasity, R. Montemanni, A. Montresor, T. Urnes, Design Patterns from Biology for Distributed Computing, TAAS 1 (2006) 26–66.

[3] J.P. Banâtre, A. Coutant, D. Le Metayer, A parallel Machine for Multiset Transformation and its Programming Style, Future Gener. Comput. Syst. 4 (1988) 133–144.

[4] J.P. Banâtre, P. Fradet, Y. Radenac, Higher-Order Chemical Programming Style, in: J.P. Banâtre, P. Fradet, J.L. Giavitto, O. Michel (Eds.), UPP, volume 3566 of *Lecture Notes in Computer Science*, Springer, 2004, pp. 84–95.

[5] J.P. Banâtre, P. Fradet, Y. Radenac, Generalised Multisets for Chemical Programming, Mathematical Structures in Computer Science 16 (2006).

[6] J.P. Banâtre, Y. Radenac, P. Fradet, Chemical Specification of Autonomic Systems, in: IASSE, ISCA, 2004, pp. 72–79.

[7] M. Bertier, M. Obrovac, C. Tedeschi, A Protocol for the Atomic Capture of Multiple Molecules on Large Scale Platforms, 13th International Conference on Distributed Computing and Networking (2012).

[8] K.M. Chandy, J. Misra, The Drinking Philosophers Problem, ACM Transactions on Programming Languages and Systems 6 (1984) 632–646.

[9] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Communications of the ACM 51 (2008) 107–113.

[10] P. Dittrich, J. Ziegler, W. Banzhaf, Artificial Chemistries – a Review, Artificial Life 7 (2001) 225–275.

[11] H. Fernandez, T. Priol, C. Tedeschi, Decentralized Approach for Execution of Composite Web Services Using the Chemical Paradigm, in: Internation Conference on Web Services, pp. 139–146.

[12] S. Grumbach, F. Wang, Netlog, a Rule-Based Language for Distributed Programming, in: M. Carro, R. Peña (Eds.), PADL, volume 5937 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 88–103.

[13] S. Hariri, M. Parashar, Handbook of Bioinspired Algorithms and Applications, Handbook of Bioinspired Algorithms and Applications, CRC Press LLC, 2005.

[14] L. Huang, W. Tong, W. Kam, Y. Sun, Implementation of GAMMA on a Massively Parallel Computer, Journal of Computer Science and Technology 12 (1997) 29–39.

[15] Z. Laliwala, R. Khosla, P. Majumdar, S. Chaudhary, Semantic and Rules Based Event-Driven Dynamic Web Services Composition for Automation of Business Processes, in: Services Computing Workshops (SCW '06), pp. 175–182.

[16] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Commun. ACM 21 (1978) 558–565.

[17] H. Lin, J. Kemp, P. Gilbert, Computing Gamma Calculus on Computer Cluster, IJTD 1 (2010) 42–52.

[18] J.W. Lloyd, Practical Advtanages of Declarative Programming, in: Joint Conference on Declarative Programming (GULP-PRODE'94), pp. 18–30.

[19] N.A. Lynch, D. Malkhi, D. Ratajczak, Atomic Data Access in Distributed Hash Tables, in: Revised Papers from the First International Workshop on Peer-to-Peer Systems, IPTPS '01, Springer-Verlag, London, UK, 2002, pp. 295–305.

[20] A. Mostéfaoui, Towards a Computing Model for Open Distributed Systems, in: V.E. Malyshkin (Ed.), PaCT, volume 4671 of *Lecture Notes in Computer Science*, Springer, 2007, pp. 74–79.

[21] MPI Forum, Message Passing Interface (MPI) Forum Home Page, `http://www.mpi-forum.org/`, 2013.

[22] C.D. Napoli, M. Giordano, J.L. Pazat, C. Wang, A Chemical Based Middleware for Workflow Instantiation and Execution, in: E.D. Nitto, R. Yahyapour (Eds.), ServiceWave, volume 6481 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 100–111.

[23] M. Parashar, S. Hariri, Autonomic Computing: An Overview, in: International Workshop on Unconventional Programming Paradigms (UPP 2004), volume 3566 of *LNCS*, Springer, Le Mont Saint-Michel, France, 2005, pp. 257–269.

[24] G. Paun, Introduction to membrane computing, in: G. Ciobanu, M.J. Pérez-Jiménez, G. Paun (Eds.), Applications of Membrane Computing, Natural Computing Series, Springer, 2006, pp. 1–42.

[25] M. Raynal, A Distributed Solution to the k-out of-M Resources Allocation Problem, Advances in Computing and Information — ICCI'91 (1991) 599–609.

[26] A. Rowstron, P. Druschel, Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems, in: Middleware 2001, volume 2218.

[27] B.A. Sanders, The Information Structure of Distributed Mutual Exclusion Algorithms, ACM Transactions on Computer Systems 5 (1987) 284–299.

[28] C. Schmidt, M. Parashar, Squid: Enabling Search in DHT-based Systems, J. Parallel Distrib. Comput. 68 (2008) 962–975.

[29] F.B. Schneider, Implementing Fault-Tolerant Services Using the State Machine Approach: a Tutorial, ACM Comput. Surv. 22 (1990).

[30] D. Skeen, M. Stonebraker, A Formal Model of Crash Recovery in a Distributed System, IEEE Transactions on Software Engineering SE-9 (1983).

[31] I. Stoica, R. Morris, D. Karger, F. Kaashoek, H. Balakrishnan, Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications, in: SIGCOMM '01, pp. 149–160.

[32] M. Viroli, F. Zambonelli, A Biochemical Approach to Adaptive Service Ecosystems, Information Sciences (2009).

[33] Y. Wang, M. Li, J. Cao, F. Tang, L. Chen, L. Cao, An ECA-Rule-Based Workflow Management Approach for Web Services Composition, in: 4th International Conference on Grid and Cooperative Computing (GCC 2005), Beijing, China, pp. 143–148.